The Developer's Guide to Containers

Using containers to maximize productivity

Andrew Hayzen | Software Engineer at KDAB

KDAB

Containers are an amazing tool for modern software development that can dramatically simplify many challenging problems. But is there more to discuss besides what we already covered in our Containers in Embedded whitepaper? Why, yes there is.

This is the quick guide to all the ways containers can improve our lives as developers. It assumes that you're already familiar with the technology through tools like docker, but please read our other whitepaper first if you need a primer.

Containers for development

What are some of the most practical uses for containers that most of us can take advantage of?

Multiple machines

If you have multiple computers (such as a desktop and laptop, or multiple laptops), containers make it trivial to move projects between machines. Instead of worrying about having all the right tools, files, or directories you need on both computers, it's easy to put the whole mess in a container. Then just copy the container to any new machine you need to work on, and you have a setup that's certain to be right every time.

I hear you saying, "I just keep all my tools in git." That might work for smaller, self-contained tools, but git isn't terribly binary friendly, and you aren't likely to create a git repository at the root of your filesystem to catch files installed into /bin, /usr/lib, or /etc. Containers let you corral big or invasive tool installations, making containers a more widely applicable solution.

Containers let you corral big or invasive tool installations, making containers a more widely applicable solution

Multiple projects

Multiple projects that share components can be a hassle. If those projects are built with different versions of shared libraries, you'll often find that you're managing conflicting dependencies unless you can bring all projects to the same baseline set. That's not always possible, especially if you're working on software that's outside of your organization's control like open-source projects or third-party components.

However, if each project puts all its files – including any dependent libraries – into a container, these projects can live side-by-side without inflicting dependency hell on its developers. This can be particularly handy if your company works on software projects for multiple clients. Every client's environment goes into a separate container, allowing clients to use whatever library, compiler, or tool chain they need without impacting any other environments the developer manages.

Experimentation

Dying to try out that new tool update? Nobody wants to find out a new package you just installed overwrote files that your project depends on, like shared libraries with a later (and API incompatible) version. Containers allow you to download and try out new software without fear that you're going to mess things up. With a properly containerized environment, it's easy to bring your computer back to exactly where you were before you started trying out experimental code.

Containers allow you to download and try out new software without fear that you're going to mess things up.

Using WINE to run Windows apps in Linux is a perfect case-in-point. A WINE installation is notorious for needing tons of assorted files. To get rid of WINE to go back to a known system state is very challenging, since you need to remove not just the app, but anything else it may have installed. Putting a WINE app into a container gives you the confidence that you can just delete the app and guarantee you're back to normal, with no dependency hangover.

Sharing

Your project is scaling up, and you need to include other developers, but it takes them more than a day to be productive just because of the development environment setup. That's why you need to put your development environment in a container. Better yet, if you keep all the team's development containers in a central registry, you can make it simple to add new members to a project, even when they're external or outsourced resources.

A containerized development environment also makes sense for small, isolated projects that don't need to scale. What if the only person who builds your company's device drivers suddenly goes to the hospital or takes parental leave? Containerized development environments can help others take over in the interim, and a well-organized corporate container registry can help your company maintain proper succession planning.

Continuous integration (CI)

A lot of tools that software engineers use during active development are overkill for the build and test environments. Why does that matter? Trimming out the fat lets you spin up more test or build instances to cut cycle times, and lets you run them on less capable machines.

Containers are the perfect solution to maximizing your resources since containers can build upon other containers to create concentric rings of capability. Leaving tools out of the development stage that doesn't need them ensures that every part of the development workflow is as efficient as it can be.

Containers are the perfect solution to maximizing your resources since containers can build upon other containers to create concentric rings of capability.

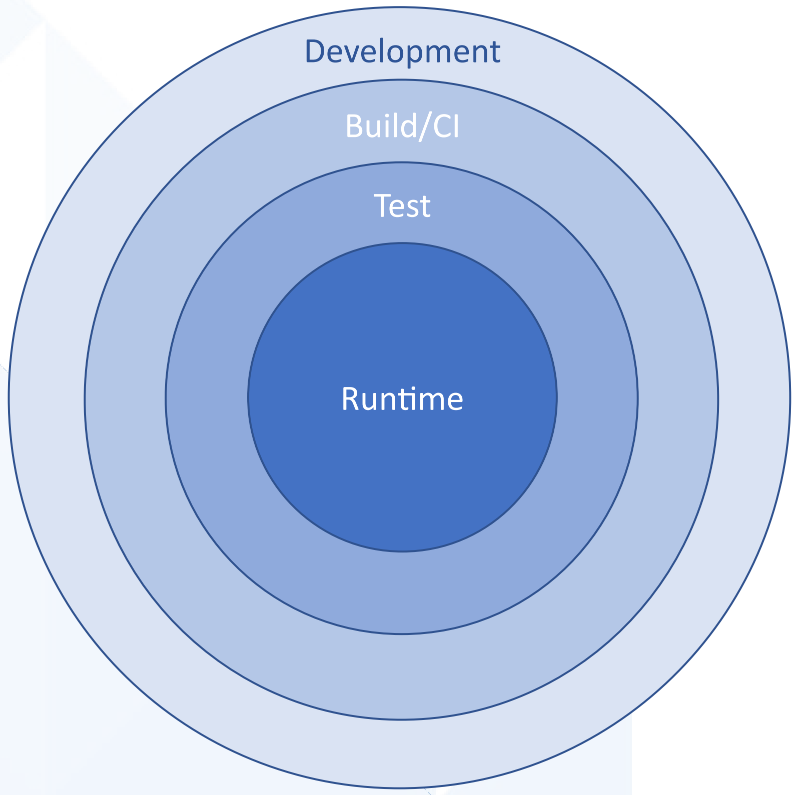

Figure 1: Using concentric containers for stages of development lets you layer on the functionality you need without duplicating the container contents.

Dependencies and documentation

Have you ever created a project and found it wouldn't build on another machine because you already had half of what you needed installed on your main development system? (Oh yeah, I forgot I needed to install Boost first ...)

Containers can help you ferret out any dependencies that you need to build or run your software. Start with a blank container and add things piece by piece until you're sure that everything you need to run the application is clearly identified without pulling in anything unnecessary. This also helps you document what pieces are needed since you need to know exactly what is going into the environment.

Containers for desktop

What specific ways can containers be used in desktop application development?

Graphical apps

Running command-line applications are a container's strength. That said, it doesn't seem possible to run GUI applications within a container. After all, containers encapsulate applications so that they are isolated from the hardware they're running on, and the screen definitely qualifies as hardware. However, by sharing X11, VNC, or Wayland sockets between the container and the host, you can run an application within a container and still use the screen.

Microservices

Although a microservice architecture is a recent trend in software organization, not all development teams are using it since it comes with some drawbacks around speed and complexity. However, one of its main benefits is scalability – something that's an ideal match with containers. Placing microservices in containers allows you to spin up as many instances as you need to handle your load.

Web

Developing web sites is a natural fit for containers since you can easily move between local interim environments to the hosted version. Create a container that emulates the web hosting environment – with your own Apache, PHP, MySQL, and other server-side tools – and your browser won't be able to tell the difference. Since the browser provides the platform integration needed to run client-side software, it doesn't need any special access outside the container.

Developing web sites is a natural fit for containers since you can easily move between local interim environments to the hosted version.

Containers for packaging

There are hundreds of Linux distributions out there, and you can't always find your favorite application for your preferred distribution. While installing programs from source is an option, it's time consuming at best. At worst, it doesn't always work if autoconf gets something wrong, if critical build files are missing, or if cross-dependencies cause the make to mysteriously fail.

Flatpak and Snap

Flatpak is a packaging system for Linux that places all the runtime requirements for an application into a container along with the application to help both Linux users and developers. By providing all dependencies along with the app, Flatpak supports executables that always run as intended on every Linux distribution. Like Flatpak, Snap provides a containerized application packaging environment that works with any Linux distribution, making it much simpler to deploy and use apps. Both Snap and Flatpak provide their own container environment through standard Linux APIs. That is, neither is built on top of docker.

By providing all dependencies along with the app, Flatpak supports executables that always run as intended on every Linux distribution.

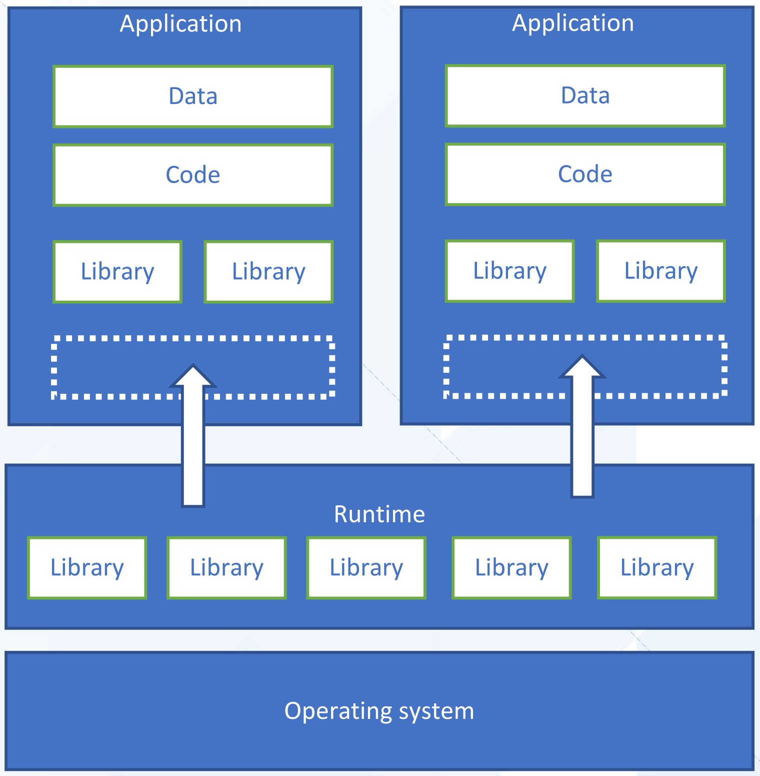

Figure 2: Flatpak architecture allows one runtime setup to be inserted into multiple apps.

What are the biggest differences between Flatpak and Snap? Snap provides a centralized and controlled app store and better supports embedded development, while Flatpak supports distributed application repositories and better supports desktop apps.

Benefits of containerized packaging

Using Snap or Flatpak means you get to build, test, and debug the same image that users are using. These environments provide already created base images that let you start with basic configurations (such as toolkits like Qt, KDE, or GNOME, and languages like C++, Python, Electron, Ruby, or Rust). They also offer application repositories – in other words, a Linux application store that users can browse. This also provides a central point to release maintenance builds.

Using Snap or Flatpak means you get to build, test, and debug the same image that users are using.

Containers inside apps

We've so far talked about containers as a developer tool and an app distribution mechanism. However, you can use container technology as part of your application implementation too.

Extra security

Let's say you want to add plug-in support to your application. If those plug-ins come from third-party developers, you might want to protect the application and your user by running plug-ins within their own containers. As an example, the Nautilus file browser for GNOME isolates the thumbnail applications that it launches to generate thumbnail images. (Nautilus thumbnailers are run using bubblewrap, which uses Linux namespace technology to implement a container-like application sandbox.)

Consistent execution

What about launcher applications? If your primary application launches many different sub-applications, what happens if two different apps use different versions of the same library?

To ensure all apps run properly, you'll need to tightly control the environment for each launched application by putting each app into a container. Game distribution systems might be an ideal candidate for this, like Valve's Steam. (Valve is currently transitioning their Steam engine to use containers for all games.)

In this case, the Steam engine itself runs normally on the host, but each game runs in a separate container to ensure it has the same environment it depended on when it was written. This lets games that are several years old still work properly, even when most of the other libraries have been updated several times in the interim. Games are also a good candidate for containerization since they primarily have screen and audio output and keyboard/mouse/joystick input – a few things that are centrally controlled.

Controlled environments

A browser-based laptop like chromebook works great for some people – families, students, and non-power-users – and can be a good way for a limited IT staff to manage many machines. However, many people might appreciate an environment that can run non-browser apps without their having to become an administrator or programmer to install or manage their system. An example is Endless OS. It lets you install apps that are based on Flatpak, and it automatically pulls the latest releases of each app, ensuring everyone has access to the best features and is using the most secure and least buggy code.

Containers are great – but they're not perfect. Let's explore some problem areas in using containers that either require workarounds, special handling, or can be showstoppers.

Problem areas for containers

Containers are great – but they're not perfect. Let's explore some problem areas in using containers that either require workarounds, special handling, or can be showstoppers.

Multi-process

Since docker runs each process in one container, applications that use multiple processes may have problems talking to each other across container boundaries. This isn't a problem for newer applications that can be built as microservices with controlled entry points. However, apps that use a wide array of inter-process communication mechanisms between processes can be difficult to convert to a containerized environment.

IDEs

Most developers use an IDE to write code, but how do you use one to develop an application that's intended to live in a container? If your IDE is something like Qt Creator, you must put it inside the container itself. There is an advantage to this, since you can then work on two (or more) applications using different versions of Qt. But the big disadvantage is the management effort and excessive size of the many Qt Creator instances installed into your application containers.

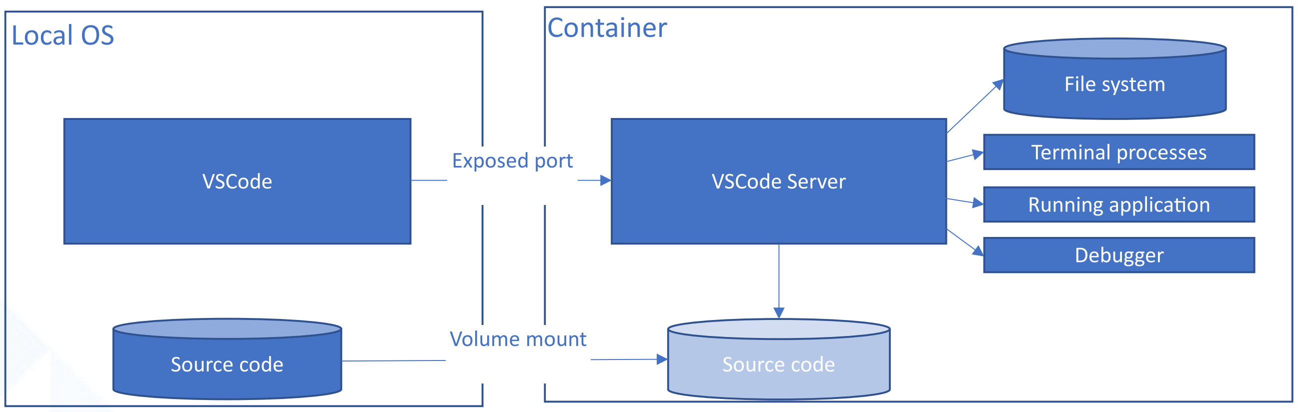

Figure 3: VSCode Remote Container architecture lets you use the full power of an IDE to develop a containerized application.

If, on the other hand, you're using VSCode, you're in luck. VSCode is very well-integrated with containers and can manage them all automatically if you install the Remote Containers plug-in. To start, you create a small json file to describe how you want to build and start your development container. From there, VSCode builds and spawns the container as needed, automatically connecting the runtime to the IDE debugging services through the container. With VSCode, you don't even realize you're running the application within a container.

```
//devcontainer.json
{
  “name”: “Node.js Sample”,
  “dockerFile”: “Dockerfile”,

  “settings”: {
    “terminal.integrated.shell.linux”: “/bin/bash”
  },

  “extensions”: [“dbaeumer.vscode-eslint”],

  “forwardPorts”: [3000],

  “postCreateCommand”: “yarn install”,

  “remoteUser”: “node”
}
```

Example 1: sample devcontainer.json gives you some simple options to use VSCode with the Remote Containers plug-in.

OSes

You may have noticed that most of our examples refer to Linux. What about Windows or MacOSX? Both OSes support docker (or other) containers. However, most of the container-based solutions are being actively developed on Linux. Partly this is because containers have a heritage that Linux shares with the cloud. It also could be because Microsoft and Apple provide their own application stores and control the development environment. As such, it becomes less valuable and more difficult to build effective containerized packaging options for them.

Regardless of the reason for a dearth of non-Linux container solutions, another problem is that containers aren't a true cross-OS technology. Unlike a virtual machine, a container depends on the underlying OS, which means it can only run containers that were built for that OS. If you want to run Linux containers, you must run them on a Linux machine or a Linux virtual machine.

Unlike a virtual machine, a container depends on the underlying OS, which means it can only run containers that were built for that OS.

Integration

Containers keep applications within their confines, which can be a problem if they need to access anything external to the container. We've already talked about how graphical applications can get around this through X11, VNC, or Wayland sockets. But what about anything else the application may need to access, such as sound, printing services, file selection dialogs, clipboards, PDF viewers, or the like?

Environments like Flatpak and Snap try to accommodate this limitation in several ways. For example, they expose D-Bus, which allows applications to send and receive messages to and from services that are running on the host machine. Both Snap and Flatpak further help application developers by incorporating xdg-desktop-portal, which uses these D-Bus messages between the container and host to provide various desktop services, such as choosing files, opening a URI, or printing.

However, any hardware resources or software services that the container or container environment doesn't explicitly support will need to have a workaround that allows the containerized application to use it. Just like xdg-desktop-portal does, these services require an in-client service that can forward requests to an authorized service running on the host that accomplishes the task. Whether this is problematic or not depends on how much of these features your application requires and whether the container in question provides a means to implement them.

Containers conclusion

Once you understand all containers have to offer, it's easy to conclude why they're considered one of the truly defining trends of the last decade. If you're trying to decide if containers make sense for you, drop us a line. We're always happy to talk software and meet new people.

About the KDAB Group

The KDAB Group is the world's leading software consultancy for architecture, development and design of Qt, C++ and OpenGL applications across desktop, embedded and mobile platforms. KDAB is the biggest independent contributor to Qt and is the world's first ISO 9001 certified Qt consulting and development company. Our experts build run-times, mix native and web technologies, solve hardware stack performance issues and porting problems for hundreds of customers, many among the Fortune 500. KDAB's tools and extensive experience in creating, debugging, profiling and porting complex applications help developers worldwide to deliver successful projects. KDAB's trainers, all full-time developers, provide market leading, hands-on, training for Qt, OpenGL and modern C++ in multiple languages.

www.kdab.com